實驗三 ARM Assembly II

Group1 0410137 劉家麟 0416324 胡安鳳

1. 實驗目的

熟悉基本 ARMv7 組合語言語法使用。

2. 實驗原理

請參考上課 Assembly 部分講義。

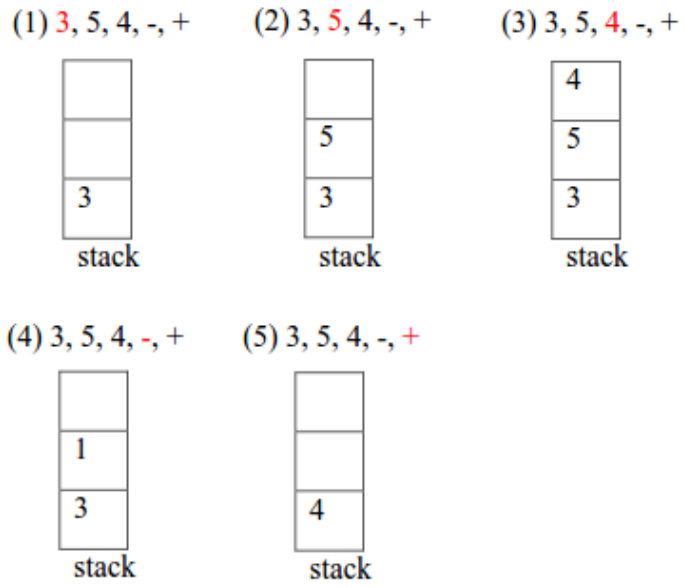
3. 實驗步驟

3.1. Postfix arithmetic

操作 stack 來完成 postfix 的加減法運算

Using stack to evaluate postfix expression which only includes addition and subtraction operations.

3.1.1. Example: 3, 5, 4, -, +



3.1.2. 實作要求

完成以下的程式碼，必須要利用 PUSH,POP 操作 stack 來完成 postfix expression 的運算，並將結果存進 expr_result 這個變數裡。

Please Complete the program below. You must use PUSH, POP operations to

calculate the result of the postfix expression, and store it into variable "expr_result".

```
   .syntax unified
   .cpu cortex-m4
   .thumb

.data
   user_stack  .zero 128
   expr_result .word   0

.text
   .global main
   postfix_expr    .asciz     "-100 10 20 + - 10 +"

main:
   LDR  R0, =postfix_expr

//TODO: Setup stack pointer to end of user_stack and calculate the
expression using PUSH, POP operators, and store the result into
expr_result

program_end:
   B    program_end

atoi:
    //TODO: implement a "convert string to integer" function
    BX LR
```

postfix_expr 格式：postfix_expr 是一串 postfix 運算式的字串，每個數字/運算子之間會用 1 個空白來區隔；input 的數字是 10 進位整數，數字正負數皆支援，字串以 ascii value 0 作為結尾；可以假設此運算式必可求出解。

Format of postfix_expr: "postfix_expr" is a postfix expression. In the expression, every operand/operator is separated with a space. The operands could be signed decimal numbers, and the operators could be "+" or "-". The string of the postfix expression is ended with a asci value 0. YOU CAN ASSUME THAT THE EXPRESSION IS LEGAL.

Prototype of atoi:

Input : start address of the string (using register)

Output : integer value (using register)

Hint:可以利用 MSR 來修改 MSP(Main Stack Pointer)的值

Hint: You can use MSR to modify the value of MSP(Main Stack Pointer)

Reference:
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0489f/CIHFIDAJ.html

http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0497a/CHDBIB

GJ.html

Note:助教會在 demo 時修改 postfix_expr 數值

Note: We will change the value of postfix_expr in demo.

1. Definition and Abstraction of the Problem

The definition and the problem can be found at http://www.geeksforgeeks.org/stack-set-4-evaluation-postfix-expression/, which is quite different from the normal infix notation.

2. Pseudo Code and Workflow of the Problem

Pseudo Code

```
get_strlen(expr[i])


for(i <length of the expression) i=string iterator
{
 if(is_integer(expr[i]))
 {
     atoi_getvalue(start_from_i);
 }
 else if(is_minus_sign)
 {

     if(is_space(get_next_char()))
     {
         //then this is the minus operation
         postfix_evaluation_minus();
     }
     else
     {
         //this is the signed part of the number
         stack_push(atoi_getvalue(start_from_i));
         //which get a minus integer
```

```
        }
    }
    else if(is_plus_sigm)
    {
        stack_push(postfix_evaluation_plus());
    }
    else //if this is a space
    {
        i++ just iterate to the next
    }
  }
```

Then we're done. Detailed code can be found at https://pastebin.com/jTQn34h7

3. Test Cases and the Results

Test Case1:

"-100 10 20 + - 10 +" should be -120

r11	0
r12	0
sp	0x2000007c
lr	134218495
pc	0x8000212 <program_end>
xPSR	1627389952
d0	0
d1	0

0x20000208 : 0x20000208 <Signed Integer>

Address	0 - 3	4 - 7	8 - B	C - F
0000000020000070	0	20	10	-120
0000000020000080	-120	0	0	536871796
0000000020000090	5368	5368	0	0
00000000200000A0	0	0	0	0
00000000200000B0	0	0	0	0

Test Case2:

"2 3 1 + + 9 -" should be -3

```
.global main
postfix_expr: .asciz  "2 3 1 + + 9 -" //
```

Console Tasks Problems Executa

	Value
sters	
	0x80001f4 (Hex)
	13 (Decimal)
	0x20000080 (Hex)
	0
	-3 (Decimal)

0x20000208 : 0x20000208 <Signed Intege

Address	0 - 3	4 - 7	8 -
0000000020000080	-3	0	0
0000000020000090	5368	5368	0
00000000200000A0	0	0	0
00000000200000B0	0	0	0
00000000200000C0	0	0	0

3.2. 求最大公因數並計算最多用了多少 stack size

在程式碼中宣告 2 個變數 m 與 n ，並撰寫 Stein 版本的最大公因數，將結果存入變數 result 裡，請使用 recursion 的寫法，並使用 stack 傳遞 function 的 parameters，禁止單純用 register 來傳。

Declare two variables “m, n”. Using Stein’s algorithm to find the GCD(Greatest Common Divisor) of them, and storing the result into variable “result”. Please use recursion to implement the algorithm and use stack to pass the parameters of the function. Don't pass the parameters with registers directly.

計算在 recursion 過程中，記錄最多用了多少 stack size，並將它存進 max_size 這個變數中。

Calculate the maximum stack size used in the recursion process, and store the result into variable “max_size”.

```
.data
   result: .word  0
   max_size:  .word  0
.text
   m: .word  0x5E
   n: .word  0x60

GCD:
    //TODO: Implement your GCD function
  BX LR
```

Prototype of GCD:

Input : A,B (using stack)

Output : GCD value (using register), max stack size (using register)

Hint: stack 的操作

Hint: manipulations of stack

```
  MOVS   R0, #1;
  MOVS R1, #2
  PUSH {R0, R1}
  LDR  R2,  [sp]     // R2 = 1
  LDR  R3, [sp, #4]  //R3 = 2
  POP     {R0, R1}
```

Note : 助教會在 demo 時修改 m, n 數值

Note: We will change the value of m, n in demo.

Reference:

GCD Algorithm（Euclid & Stein）：

http://www.cnblogs.com/drizzlecrj/archive/2007/09/14/892340.html

1. Definition and Abstraction of the Problem

The steps to find GCD using Stein's Algorithm gcd(a, b).

(1) If both a and b are 0s, gcd is zero gcd(0, 0)=0.

(2) gcd(a, 0) = gcd(0, b) = 0, because every number divides 0.

(3) If a and b are both even, gcd(a, b) = 2*gcd(a/2, b/2) because 2 is a common divisor. Multiplication with 2 can be done with bitwise shift operator.

(4) If a is even and b is odd, gcd(a, b) = gcd(a/2, b). Similarly, if a is odd and b is even, then gcd(a, b) = gcd(a, b/2). It is because 2 is not a common divisor.

(5) If both a and b are odd, then gcd(a, b) = gcd(|a-b|/2, b). Note that difference of two odd numbers is even.

(6) Repeat steps 3–5 until a = b, or until a = 0. In either case, the GCD is power(2, k) * b, where power(2, k) is 2 raise to the power of k and k is the number of common factors of 2 found in step 2.

Implementation in C code (Recursive)

```
int gcd(int a, int b)
{
    if (a == b)
        return a;

    /* GCD(0,b) == b; GCD(a,0) == a, GCD(0,0) == 0 */
    if (a == 0)
        return b;
    if (b == 0)
        return a;

    // look for factors of 2
    if (~a & 1 )        // a is even
    {
        if (b & 1)      // b is odd
            return gcd(a >> 1, b);
        else            // both a and b are even
            return gcd(a >> 1, b >> 1) << 1;
```

```
        }

        if (~b & 1)         // a is odd, b is even
            return gcd(a, b >> 1);

        // reduce larger number
        if (a > b)
            return gcd((a - b) >> 1, b);

        return gcd((b - a) >> 1, a);
    }
```

Reference: http://www.geeksforgeeks.org/steins-algorithm-for-finding-gcd/

2. Pseudo Code and Workflow of the Problem

Pseudo Code of Assembly

```
main:
    push m and n into stack
    branch to GCD
    store the result and max_size back to memory
```

```
GCD:
    Load m and n from stack
    If m==n → return m
    If m==0 → return n
    If n==0 → return m
    If m is even → m is even
    If n is even && m is odd → n is even
    If both m and n are odds → m>n or m<n
    Return m:
        Move m to result register
        Branch
    Return n:
        Move n to result register
```

```
        Branch
    M is even:
        If n is even → both even
        Else recursive → GCD(m>>1,n)
    Both even:
        Recursive → gcd(m>>1, n>>1)<<1
    N is even:
        Recursive → gcd(m,n>>1)
    M>N:
        Recursive → gcd((m-n)>>1,n)
    N>M:
        Recursive → gcd(m,(n-m)>>1)
    Recursive:
        Push m(r0), n(r1) into stack just like main
        Pop out lr, so that recursion can back to right address
```

Bx lr can only save the address where we are going to go once, so when we implement the recursive function call, we should be careful of the right lr address.

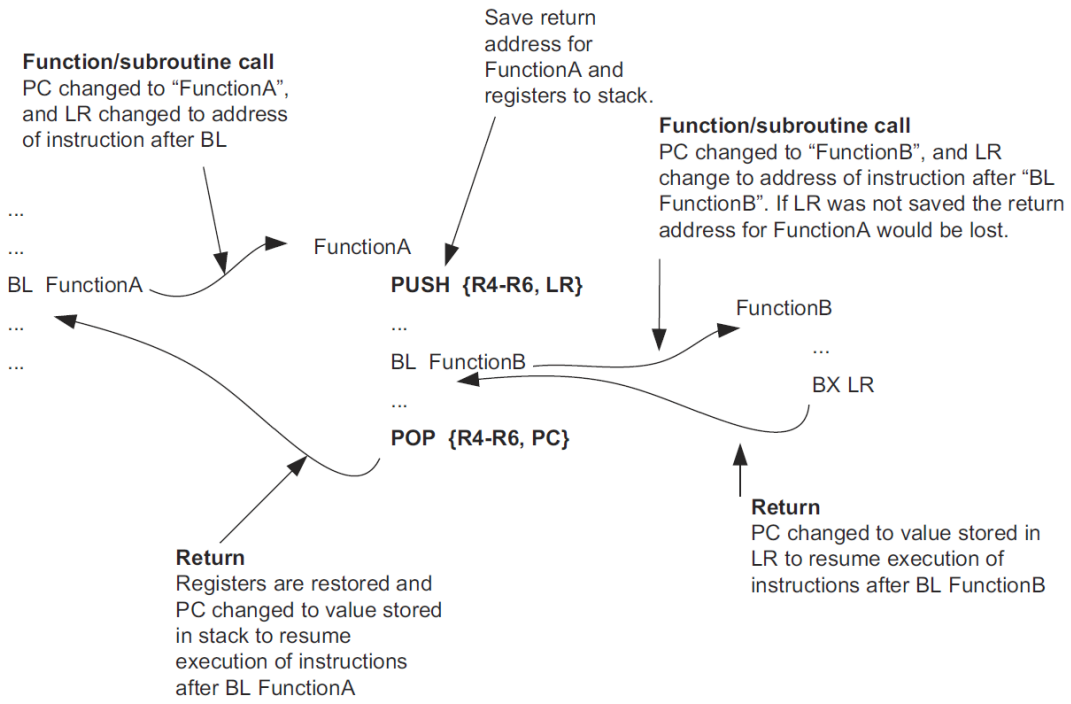


Figure 6.3:
Using push and pop of multiple registers in functions.

(Reference: TA's powerpoint)

To solve the problem of bl and bx lr, we use stack to push in new lr if we get into a new recursive loop, and pop out lr when leaving the loop.

3. Test Cases and the Results

Test Case1:

m = 0x5E (94)
n = 0x60 (96)
gcd(m,n) = 2

Name	Value
General Registers	
r0	0x20000000 (Hex)
r1	0x20000004 (Hex)
r2	0x1 (Hex)
r3	1 (Decimal)
r4	1
r5	0
r6	1 (Decimal)
r7	2
r8	12

0x20000000 : 0x20000000 <Hex> New Renderings...

Address	0 - 3	4 - 7	8 - B	C - F
20000000	02000000	0C000000	00000000	F4020020
20000010	5C030020	C4030020	00000000	00000000
20000020	00000000	00000000	00000000	00000000
20000030	00000000	00000000	00000000	80040008
20000040	00000000	00000000	00000000	00000000
20000050	00000000	00000000	00000000	00000000
20000060	00000000	00000000	00000000	00000000

max_size = 12

Test Case2:

m = 0x19 (25)
n = 0x208 (520)
gcd(m,n) = 5

Name	Value
General Registers	
r0	0x20000000 (Hex)
r1	0x20000004 (Hex)
r2	0x5 (Hex)
r3	5 (Decimal)
r4	1
r5	0
r6	0 (Decimal)
r7	5
r8	9

0x20000000 : 0x20000000 <Signed Integer> New Renderings...

Address	0 - 3	4 - 7	8 - B	C - F
20000000	5	9	0	536871668
20000010	536871772	536871876	0	0
20000020	0	0	0	0
20000030	0	0	0	134218880
20000040	0	0	0	0
20000050	0	0	0	0
20000060	0	0	0	0

max_size = 9

Test Case3:

m = 0x8A (138)

n = 0x4D (77)

gcd(m,n) = 1

Name	Value
General Registers	
r0	0x20000000 (Hex)
r1	0x20000004 (Hex)
r2	0x1 (Hex)
r3	1 (Decimal)
r4	1
r5	0
r6	0 (Decimal)
r7	1
r8	11

0x20000000 : 0x20000000 <Signed Integer> New Renderings...

Address	0 - 3	4 - 7	8 - B	C - F
20000000	1	11	0	536871668
20000010	536871772	536871876	0	0
20000020	0	0	0	0
20000030	0	0	0	134218880
20000040	0	0	0	0
20000050	0	0	0	0

max_size = 11

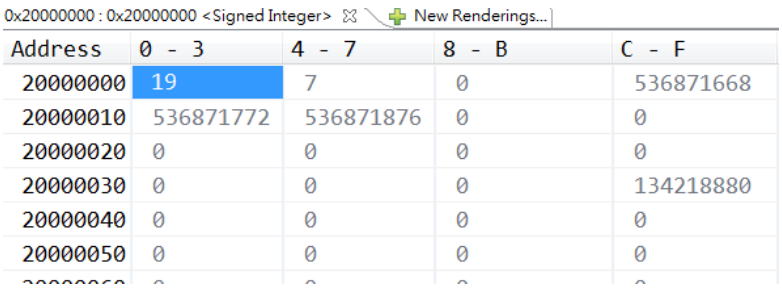
Test Case4:

m = 0x39 (57)

n = 0x260 (608)

gcd(m,n) = 19

Name	Value
General Registers	
r0	0x20000000 (Hex)
r1	0x20000004 (Hex)
r2	0x13 (Hex)
r3	19 (Decimal)
r4	1
r5	0
r6	0 (Decimal)
r7	19
r8	7



max_size = 7

Test Case5:

m = 0x400 (1024)

n = 0x200 (512)

gcd(m,n) = 512

```
.text
    m: .word 0x200 //94
    n: .word 0x400 //96
    .global main
```